Défonctionnaliser pour prouver

Mário Pereira

Défonctionnaliser pour prouver

Mário Pereira[1] *

1: Lab. de Recherche en Informatique,
Univ. Paris-Saclay, CNRS, Orsay, F-91405
et Inria Saclay – Île-de-France, Orsay, F-91893
Mario.Parreira-Pereira@lri.fr

Résumé

Cet article explore l'idée d'utiliser la défonctionnalisation comme une technique de preuve de programmes d'ordre supérieur. La défonctionnalisation consiste à remplacer les valeurs fonctionnelles par une représentation du premier ordre. L'intérêt est alors de pouvoir utiliser ensuite un outil de preuve de programmes existant, sans lui ajouter de support pour l'ordre supérieur. Cet article illustre et discute cette approche à l'aide de plusieurs exemples et de l'outil de vérification déductive Why3.

1. Introduction

Un programme d'ordre supérieur est un programme qui utilise les fonctions comme des valeurs de première classe. Dans un tel programme, les fonctions peuvent être passées comme des arguments à d'autres fonctions ou renvoyées comme le résultat d'un certain calcul. Le concept d'ordre supérieur est largement connu et utilisé dans les langages dits *fonctionnels*, tels que OCaml, Haskell ou SML. Plus récemment des langages tels que Java ou C++ ont introduit un support pour des fonctions d'ordre supérieur.

La preuve de correction de programmes d'ordre supérieur pose des défis complexes, en particulier dans le cadre de programmes avec des effets de bord. Certaines méthodologies existantes [2,14] utilisent des assistants de preuve interactifs et un encodage des effets directement dans la logique de l'assistant de preuve. Dans le contexte de la preuve automatique de programmes, Kanig et Filliâtre [12] ont proposé un système de types et effets et un calcul de plus faibles préconditions pour spécifier et prouver la correction fonctionnelle de programmes d'ordre supérieur. Leur système semble néanmoins difficile à utiliser en pratique.

Dans cet article, on propose une nouvelle méthodologie pour la vérification de programmes d'ordre supérieur qui comportent potentiellement des effets. On explore l'idée d'utiliser la technique de défonctionnalisation pour prouver de tels programmes. L'utilisation de la défonctionnalisation nous permet d'obtenir un programme du premier ordre équivalent au programme d'ordre supérieur original, ce qui nous ramène à un contexte de preuve de programmes du premier ordre. De cette façon, on peut utiliser des outils de preuve existants, sans avoir besoin de les étendre avec un support pour l'ordre supérieur. Cette technique présente comme limitation le fait qu'on doit connaître à l'avance toutes les fonctions qui seront utilisées comme des valeurs de première classe. Même si cela nous empêche d'appliquer cette technique pour prouver n'importe quel programme d'ordre supérieur, il

∗. Ce travail est en partie soutenu par la Fondation des Sciences et Technologie Portugaise (bourse FCT-SFRH/BD/99432/2014) et par l'Agence National de Recherche Française (projet VOCAL ANR-15-CE25-0008).

semble exister, néanmoins, un ensemble intéressant et représentatif de programmes d'ordre supérieur auxquels on peut appliquer la défonctionnalisation pour prouver leur correction. Nous présentons notre approche à l'aide de plusieurs exemples écrits et vérifiés avec le système de preuve de programmes Why3 [7].

Le reste de l'article est organisé comme suit. La section 2 introduit la technique de défonctionnalisation à l'aide d'un exemple. Dans la section 3 on explique en détail notre proposition d'utilisation de la défonctionnalisation comme un moyen de preuve. On illustre notre approche avec plusieurs exemples. L'article s'achève sur quelques perspectives et conclusions sur notre travail. Le développement Why3 présenté dans cet article peut être trouvé à l'adresse suivante : `http://www.lri.fr/~mpereira/defunc`.

2. Défonctionnalisation

La défonctionnalisation est une technique de transformation des programmes d'ordre supérieur vers des programmes du premier ordre. Elle a été introduite par Reynolds [18] comme un moyen d'obtenir un interpréteur du premier ordre à partir d'un interpréteur d'ordre supérieur. Plus récemment, cette technique a été largement étudiée et utilisée par Danvy et ses étudiants [3, 4]. En particulier, ils ont montré comment on peut dériver des machines abstraites pour différentes stratégies d'évaluation du lambda-calcul à partir d'interpréteurs compositionnels [1].

Expliquons le principe de la défonctionnalisation à l'aide d'un exemple écrit en OCaml [1]. Considérons le programme qui calcule la hauteur d'un arbre binaire, écrit en style CPS [15] :

```
type 'a tree = E | N of 'a tree * 'a * 'a tree

let rec heigth_tree_cps (t: 'a tree) (k: int → 'b) : 'b = match t with
  | E → k 0
  | N (l, x, r) →
     heigth_tree_cps l (fun hl →
     heigth_tree_cps r (fun hr → k (1 + max hl hr)))

let heigth_tree t = heigth_tree_cps t (fun x → x)
```

Ce programme à été sciemment écrit en style CPS pour éviter tout débordement de pile (*stack overflow*), quelle que soit la forme de l'arbre. La transformation CPS est un moyen mécanique pour éviter les débordements de pile, dès lors que le compilateur optimise l'appel terminal.

Remarquons la présence, dans la fonction `heigth_tree_cps`, de l'argument `k` du type `int → 'b`. C'est l'utilisation de cet argument qui donne à `heigth_tree_cps` son caractère d'ordre supérieur. Cet argument joue le rôle d'une *continuation*, c'est-à-dire qu'avant de renvoyer son résultat, la fonction `heigth_tree_cps` le passe à `k`. Dans le cas d'un arbre vide (première branche du filtrage), par exemple, on applique `k` à 0 au lieu de renvoyer directement ce résultat.

Notons les trois fonctions anonymes utilisées dans le code de ce programme. Dans l'appel `heigth_tree_cps l` la continuation `(fun hl → ...)` est appliquée au résultat du calcul de la hauteur du sous-arbre gauche `l`. L'argument `hl` représente la hauteur de `l` déjà calculée. À l'intérieur de cette continuation on trouve un deuxième appel récursif avec une autre continuation, cette fois pour le sous-arbre droit `r`. Au moment d'appliquer la fonction anonyme `(fun hr → ...)` on connaît déjà tous les ingrédients nécessaires pour calculer la hauteur de l'arbre initial `t`. Encore une fois, on ne renvoie pas directement le résultat `1 + max hl hr`, mais on le passe à `k`. La troisième fonction anonyme trouvée dans ce programme est la fonction identité `(fun x → x)`. L'application de cette fonction garantit que le résultat renvoyé par `heigth_tree t` est bien la hauteur de l'arbre `t`.

1. Ce code peut être facilement adapté à n'importe quel langage qui supporte les fonctions comme valeurs de première classe.

```
type 'a tree = E | N of 'a tree * 'a * 'a tree

type 'a kont = Kid | Kleft of 'a tree * 'a kont | Kright of 'a kont * int

let rec apply (k: 'a kont) arg = match k with
  | Kid → let x = arg in x
  | Kleft (r, k) → let hl = arg in heigth_tree_cps r (Kright (k, hl))
  | Kright (k, hl) → let hr = arg in apply k (1 + max hl hr)
and heigth_tree_cps t (k: 'a kont) = match t with
  | E → apply k 0
  | N (l, x, r) → heigth_tree_cps l (Kleft (r, k))

let heigth_tree t = heigth_tree_cps t Kid
```

FIGURE 1 – Programme qui calcule la hauteur d'un arbre, défonctionnalisé.

Pour défonctionnaliser ce programme, il nous faut une représentation du premier ordre pour les trois abstractions utilisées. L'idée est de remplacer le type fonctionnel par un type algébrique, dans lequel on capture les valeurs des variables libres utilisées dans chaque fonction. Pour l'exemple ci-dessus, nous introduisons le type suivant :

```
type 'a kont = Kid | Kleft of 'a tree * 'a | Kright of 'a kont * int
```

Le constructeur `Kid` représente la fonction identité, et donc il n'y a pas de variables libres à capturer. Les constructeurs `Kleft` et `Kright` représentent, respectivement, les fonctions `(fun hl → ...)` et `(fun hr → ...)`. Leurs arguments capturent les variables libres utilisées dans chacune de ces deux fonctions. Dans le cas de `Kleft` on garde le sous-arbre `r` et la continuation `k` ; dans le cas de `Kright` le premier argument est la valeur de `k` et le deuxième représente `hl`, la hauteur du sous-arbre gauche.

Ayant entre les mains une représentation du premier ordre pour les abstractions, remplaçons toutes les abstractions par le constructeur de `'a kont` correspondants :

```
let rec heigth_tree_cps t (k: 'a kont) = match t with
  | E → ??? (* application de k défonctionnalisé à son argument *)
  | N (l, x, r) → heigth_tree_cps l (Kleft (r, k))

let heigth_tree t = heigth_tree_cps t Kid
```

La deuxième étape du processus de défonctionnalisation consiste à introduire une fonction `apply` pour remplacer les applications dans le programme de départ. Cette fonction prend en argument une valeur du type `kont 'a` sur laquelle elle va réaliser une analyse par cas. Selon le constructeur, la fonction `apply` exécute le code de l'abstraction qui correspond à ce constructeur. L'argument de l'application est passé à `apply` comme un second argument. La fonction `apply` pour notre exemple est la suivante :

```
let rec apply (k: 'a kont) arg = match k with
  | Kid → let x = arg in x
  | Kleft (r, k) → let hl = arg in heigth_tree_cps r (Kright (k, hl))
  | Kright (k, hl) → let hr = arg in apply k (1 + max hl hr)
```

Pour obtenir le programme complètement défonctionnalisé, il suffit de remplacer toutes les applications de la continuation `k` par des appels à la fonction `apply`. La figure 1 contient le code complet pour cet exemple.

3. Preuve par défonctionnalisation

Dans cette section on explore l'idée d'utiliser la défonctionnalisation comme une technique de preuve pour des programmes d'ordre supérieur avec effets. Notre proposition est la suivante :

1. étant donné un programme d'ordre supérieur (possiblement avec effets), on lui ajoute une spécification logique.
2. on défonctionnalise le programme et on traduit en même temps sa spécification pour qu'elle devienne une spécification du programme défonctionnalisé.
3. on utilise un outil de preuve de programmes existant pour prouver le programme défonctionnalisé.

On illustre cette proposition sur plusieurs exemples : le programme qui calcule la hauteur d'un arbre (section 3.1) ; un programme qui calcule le nombre d'éléments distincts d'un arbre (section 3.2) ; un interpréteur à petits pas pour un petit langage (section 3.3).

Nos expériences sont réalisées à l'aide de l'outil de vérification déductive Why3. Comme à l'heure actuelle Why3 ne permet pas de raisonner sur des programmes d'ordre supérieur, tous les exemples portant sur la spécification et preuve de programmes d'ordre supérieur sont écrits dans un système hypothétique qu'on peut voir comme une extension de Why3.

3.1. Hauteur d'un arbre

Reprenons l'exemple de la section 2. Pour spécifier ce programme, on doit fournir des contrats pour les fonctions `heigth_tree_cps` et `heigth_tree`. La fonction `heigth_tree` renvoie la hauteur de l'arbre `t` passé en argument, comme on le spécifie dans sa postcondition :

```
let heigth_tree (t: tree 'a) : int
  ensures { result = height t }
= heigth_tree_cps t (fun x → x)
```

Ici `result` est un mot-clé de Why3 pour représenter la valeur renvoyée et `height` est une fonction logique qui donne la hauteur d'un arbre.

La valeur renvoyée par la fonction `height_tree_cps` est le résultat de l'application de la continuation `k` à la hauteur de l'arbre `t`. Ce serait naturel de donner à `height_tree_cps` la postcondition suivante :

```
ensures { result = k (height t) }
```

Cependant, cette spécification pose le problème d'interpréter l'utilisation de fonctions du programme dans la logique. En toute généralité, une telle utilisation peut facilement produire des incohérences logiques, puisque les fonctions du programme peuvent avoir des effets ou ne pas terminer. On fait alors le choix de restreindre notre langage de spécification : on s'autorise à utiliser des noms des fonctions du programme, mais on impose une barrière d'abstraction entre le monde logique et le programme. Pour cela, on adopte un système dans lequel les fonctions sont abstraites par une paire de prédicats qui représente leur précondition et postcondition [17]. Ainsi, à l'intérieur d'une formule logique, une fonction f de type $\tau_1 \to \tau_2$ devient une paire de prédicats :

$$\begin{aligned} \texttt{pre}\ f : &\quad \tau_1 \to \texttt{prop} \\ \texttt{post}\ f : &\quad \tau_1 \to \tau_2 \to \texttt{prop} \end{aligned}$$

On utilise `pre` f et `post` f pour faire référence à la précondition et à la postcondition de f, respectivement.

Revenons à l'exemple de la hauteur d'un arbre. Nous pouvons donc écrire, avec la notation ci-dessus, le contrat suivant pour la fonction `height_tree_cps` :

```
let rec height_tree_cps (t: tree 'a) (k: int → 'b) : 'b
  ensures { post k (height t) result }
```

Cette postconditon impose une relation entre la valeur qui est passée à `k` (la hauteur de `t`) et le résultat renvoyé (`result`). En suivant cette méthode, on peut aussi donner des contrats aux fonctions anonymes utilisées à l'intérieur de `height_tree_cps` :

```
let rec heigth_tree_cps (t: tree 'a) (k: int → 'b) : 'b
  ensures { post k (heigth t) result }
= match t with
  | Empty → k 0
  | Node l x r →
     heigth_tree l (fun hl → ensures { post k (1 + max hl (height r)) result }
     heigth_tree r (fun hr → ensures { post k (1 + max hl hr) result }
       k (1 + max hl hr)))
  end
```

Pour la première abstraction, on spécifie que le résultat de son application est en relation avec la hauteur de l'arbre, en utilisant la hauteur `hl` du sous-arbre gauche et la hauteur du sous-arbre droit, qu'on n'a pas encore calculée. On ajoute à la deuxième abstraction une postcondition similaire, à ceci près qu'au moment d'appliquer cette fonction on connaît déjà la hauteur `hr`.

On va maintenant défonctionnaliser ce programme, avec sa spécification, pour ensuite le prouver à l'aide de l'outil Why3. Le code obtenu est le suivant :

```
type kont 'a = Kid | Kleft (tree 'a) (kont 'a) | Kright int (kont 'a)

let rec heigth_tree_cps (t: tree 'a) (k: kont 'a) : int
  ensures { post k (height t) result }
= match t with
  | Empty → apply k 0
  | Node l _ r → heigth_tree_cps l (Kleft r k)
  end
with apply (k: kont 'a) (arg: int) : int
  ensures  { post k arg result }
= match k with
  | Kid → arg
  | Kleft r k → heigth_tree_cps r (Kright arg k)
  | Kright hl k → apply k (1 + max hl arg)
  end

let height_tree (t: tree 'a) : int
  ensures { result = height t }
= heigth_tree_cps t Kid
```

On peut noter qu'il n'y a pas de différence avec le programme OCaml de la figure 1, si ce n'est la syntaxe de Why3 et les annotations logiques. Notamment, les fonctions `heigth_tree_cps` et `apply` restent quand même des fonctions mutuellement récursives.

La spécification du programme défonctionnalisé est la même que celle du programme de départ. En particulier, on a gardé notre utilisation de la projection `post` pour la postcondition de `heigth_tree_cps`. Il nous faut aussi donner une spécification à la fonction `apply`, qui est nouvelle. Comme la fonction `apply` simule l'application d'une fonction à son argument, la seule spécification qu'on peut lui donner est que sa postcondition est bien la postcondition de la fonction `k`. De même, la précondition pour `apply` serait la précondition de `k` et on pourrait y accéder en utilisant la projection `pre`. On a choisit de ne pas le faire, vu qu'il s'agit ici de la précondition triviale.

Enfin, il nous faut introduire le prédicat `post` dans notre spécification logique. Pour faire cela,

on crée un prédicat `post` qui rassemble les postconditions fournies dans le programme de départ. Un peu comme pour la fonction `apply`, ce prédicat effectue un filtrage sur le type algébrique `kont 'a` et pour chaque constructeur on copie la postcondition donnée dans l'abstraction correspondante. Ici, le prédicat `post` est le suivant :

```
predicate post (k: kont 'a) (arg result: int) = match k with
 | Kid → let x = arg in x = result
 | Kleft r k' → let hl = arg in post k' (1 + max hl (height r)) result
 | Kright hl k' → let hr = arg in post k' (1 + max hl hr) result
 end
```

Pour la postcondition du constructeur `Kid` on utilise la postcondition triviale `result = x`. Cette formule est la postcondition la plus forte de cette fonction ; on pourrait ici l'inférer automatiquement. Quand on passe ce programme à Why3 toutes les obligations de preuve générées sont automatiquement prouvées en utilisant des démonstrateurs SMT.

Nous voulons faire ici une remarque importante : la façon dont nous avons défonctionnalisé notre programme et sa spécification, en particulier la façon de construire le prédicat `post` et le contrat pour la fonction `apply`, peut être mécanisée. Nous pourrions tout à fait imaginer un outil qui prend un programme d'ordre supérieur annoté, qui le défonctionnalise et qui l'envoie ensuite à l'outil Why3.

Terminaison. Un aspect important qui n'a pas été traité dans cet exemple est la preuve de terminaison. Il est possible de prouver que le programme défonctionnalisé termine en le spécifiant avec des mesures appropriées. Pour cela, on introduit des fonctions logiques pour compter le nombre de nœuds d'un arbre et des constructeurs du type `kont 'a`. Ces mesures sont assez subtiles et nécessitent un peu d'imagination :

```
function var_tree (t: tree 'a) : int = match t with
  | Empty → 1
  | Node l _ r → 3 + var_tree l + var_tree r
  end

function var_kont (k: kont 'a) : int = match k with
  | Kid → 0
  | Kleft r k → 2 + var_tree r + var_kont k
  | Kright _ k → 1 + var_kont k
  end
```

On prouve qu'effectivement ces fonctions peuvent être utilisées comme des mesures de décroissance, vu qu'elles ne dépassent jamais une valeur minimum :

```
lemma var_tree_nonneg: forall t: tree 'a. var_tree t ≥ 0

lemma var_kont_k_nonneg: forall k: kont 'a. var_kont k ≥ 0
```

Il nous reste à ajouter des annotations de terminaison appropriées dans notre programme :

```
let rec height (t: tree 'a) (k: kont 'a) : int
  variant { var_tree t + var_kont k }
  ...
with apply (k: kont 'a) (arg: int) : int
  variant { var_kont k }
  ...
```

Toutes les obligations de preuve générées concernant la terminaison sont aussi automatiquement prouvées.

Il serait intéressant d'avoir un mécanisme pour écrire une spécification sur la terminaison d'un programme d'ordre supérieur et la traduire automatiquement, tel qu'on l'a fait pour le prédicat `post`.

3.2. Nombre d'éléments distincts dans un arbre

Le prochain exemple est celui du calcul du nombre d'éléments distincts d'un arbre binaire. On adopte un style CPS pour éviter les débordements de pile. Ce programme diffère de celui de la section précédente par la présence d'effets de bord. On utilise un ensemble mutable (une référence vers un ensemble fini) pour stocker les éléments déjà vus. À la fin du programme, on renvoie le cardinal de cet ensemble et on obtient donc le nombre d'éléments distincts d'un arbre. Voici le code Why3 :

```
let n_distinct_elements (t: tree 'a) : int =
  let h = ref empty in
  let rec distinct_elements_loop (t: tree 'a) (k: unit → unit) : unit =
    match t with
    | Empty → k ()
    | Node l x r →
       h := add x !h;
       distinct_elements_loop l (fun () →
       distinct_elements_loop r (fun () → k ()))
    end
  in
  distinct_elements_loop t (fun x → x);
  cardinal !h
```

Les opérations sur les ensembles utilisées dans ce programme sont issues de la bibliothèque standard de Why3. Les continuations présentes dans ce programme sont utilisées de manière similaire à celles du programme de la section 3.1. On traite le sous-arbre gauche dans l'appel `distinct_elements_loop l (fun () → ...)` ; `distinct_elements_loop r (fun () → ...)` est utilisé pour traiter le sous-arbre ; pour assurer qu'on renvoie le résultat correct, le premier appel à `distinct_elements_loop` depuis `distinct_elements` est fait avec la fonction identité.

Étant écrit en style CPS, ce programme combine l'utilisation de l'ordre supérieur avec des effets de bord. On doit donc prendre en compte la notion d'état du programme dans sa spécification. Pour faire cela, on modifie la représentation des fonctions au niveau logique, comme dans la thèse de J. Kanig [11]. On introduit d'abord un nouveau type `state` et on étend le type des fonctions dans la logique comme suit :

$$\begin{aligned} \texttt{pre}\ f : &\quad \tau_1 \to \texttt{state} \to \texttt{prop} \\ \texttt{post}\ f : &\quad \tau_1 \to \texttt{state} \to \texttt{state} \to \tau_2 \to \texttt{prop} \end{aligned}$$

L'argument supplémentaire de la précondition correspond à l'état au moment de l'appel de la fonction. Les deux arguments supplémentaires de la postcondition représentent, respectivement, l'état *avant* et *après* l'exécution de la fonction. La nature du type `state` dépendra des fonctions considérées. Pour cet exemple, l'état est réduit à `set 'a`.

Revenons au code de la fonction `n_distinct_elements`. Pour spécifier `n_distinct_elements` et `distinct_elements_loop` on introduit d'abord une fonction logique `set_of_tree`. Cette fonction calcule l'union de l'ensemble des éléments d'un arbre avec un ensemble `s` donné, qui joue le rôle d'accumulateur :

```
function set_of_tree (t: tree 'a) (s: set 'a) : set 'a = match t with
  | Empty → s
  | Node l x r → set_of_tree r (set_of_tree l (add x s))
  end
```

Pour calculer l'ensemble des éléments d'un arbre, il suffit d'appeler `set_of_tree` avec l'ensemble vide comme second argument. On donne le contrat suivant à la fonction `n_distinct_elements` :

```
let n_distinct_elements (t: tree 'a) : int
  ensures { result = cardinal (set_of_tree t empty) }
```

On spécifie `distinct_elements_loop` et les continuations utilisées de façon similaire :

```
let rec distinct_elements_loop (t: tree 'a) (k: unit → unit) : unit
  ensures { post k () (set_of_tree t (old !h)) !h () }
= match t with
  | Empty → k ()
  | Node l x r →
     h := add x !h;
     distinct_elements_loop l (fun () → ensures { post k () (set_of_tree r (old !h)) !h () }
     distinct_elements_loop r (fun () → ensures { post k () (old !h) !h () }
       k ()))
  end
```

La postcondition de `distinct_elements_loop` mérite une explication détaillée. Due au fait qu'on utilise la continuation `k` à l'intérieur de `distinct_elements_loop`, la postcondition de cette fonction dépend de la postcondition de `k`. Il faut alors caractériser l'état du programme quand on appelle `k` et l'état après son exécution. Ce dernier est l'état après l'exécution toute entière de la fonction `distinct_elements_loop`, représenté par la valeur contenue dans la référence `h`. L'état initial est plus subtil. Rappelons-nous qu'au moment d'appliquer `k` ou aura déjà parcouru tout l'arbre `t`. Alors, l'état avec lequel on appelle `k` est l'ensemble de tous les éléments de `t` réunis avec la valeur contenue dans `h` au moment de l'appel à `distinct_elements_loop`. On récupère cette valeur en utilisant l'étiquette `old` de Why3, ce qui nous permet d'accéder à la valeur contenue dans une référence à l'entrée d'une fonction.

La spécification des continuations utilisées à l'intérieur des appels récursifs à `distinct_elements_loop` suit le raisonnement qu'on vient de décrire. Les postconditions de ces deux abstractions dépendent aussi de la postcondition de `k`. Dans l'appel `distinct_elements_loop l`, on spécifie dans la postcondition de sa continuation que l'état avec lequel on appellera la continuation est `set_of_tree r (old !h)`. Ici, `old !h` représente l'état avant d'appeler la continuation, c'est-à-dire après avoir parcouru le sous-arbre `l`. Comme cette continuation est utilisée pour parcourir tout le sous-arbre droit, alors au moment d'appliquer `k` la référence `h` contiendra l'ensemble des éléments distincts de `r` réunis avec l'ensemble des éléments distincts de `l` et `x` ajouté initialement. Finalement, pour `distinct_elements_loop r`, sa continuation ne fait qu'appeler `k` et on fait donc cet appel avec l'état initial `old !h`, qui est l'état exactement avant d'appeler `k`.

Pour défonctionnaliser ce programme et traduire sa spécification, on commence par introduire le type algébrique qui représente les continuations :

```
type kont 'a = Kid | Kleft (tree 'a) (kont 'a) | Kright (kont 'a)
```

Les seules effets produits dans cet exemple sont les accès (en écriture et lecture) à la référence `h`. Ainsi, on peut définir le type `state` comme étant le type des valeurs pointées par `h` :

```
type state 'a = set 'a
```

Finalement, on introduit le prédicat `post` :

```
predicate post (k: kont 'a) (arg: unit) (old cur: state 'a) (result: unit)
= match k with
  | Kid → let () = arg in old == cur
  | Kleft r k → let () = arg in post k () (set_of_tree r old) cur result
  | Kright k → let () = arg in post k () old cur result
  end
```

Pour le cas de la fonction identité, ce prédicat spécifie que l'état de sortie de la fonction est le même qu'à l'entrée. Le symbole `(==)` représente l'égalité extensionnelle de deux ensembles. En utilisant cette définition de `post`, on produit la fonction `apply` avec sa spécification. On arrive ainsi au code suivant :

```
let n_distinct_elements (t: tree 'a) : int
```

```
  ensures { result = cardinal (set_of_tree t empty) }
= let h = ref empty in
  let rec apply (k: kont 'a) (arg: unit) : unit
    ensures { post k arg (old !h) !h () }
  = match k with
    | Kid → let x = arg in x
    | Kleft r k → let _ = arg in distinct_elements_loop r (Kright k)
    | Kright k → let _ = arg in apply k arg
    end
  with distinct_elements_loop (t: tree 'a) (k: kont 'a) : unit
    ensures { post k () (set_of_tree t (old !h)) !h () }
  = match t with
    | Empty → apply k ()
    | Node l x r →
       h := add x !h;
       distinct_elements_loop l (Kleft r k)
    end
  in
  distinct_elements_loop t Kid;
  cardinal !h
```

Une fois passé à Why3, toutes les obligations de preuve engendrées pour ce programme sont automatiquement prouvées. La terminaison de ce programme pourrait aussi être prouvée, en utilisant des arguments de terminaison identiques à ceux qu'on a utilisés pour l'exemple précédent. En fait, les mesures de décroissance introduites pour le programme de la section précédente pourraient aussi être utilisées pour ce programme, à la différence près des deux types `kont`.

3.3. Interprète à petits pas

Le dernier exemple qu'on présente est celui d'un interpréteur à petits pas pour un mini langage d'expressions arithmétiques. Il s'agit d'un langage limité aux littéraux et à des soustractions :

```
type exp = Const int | Sub exp exp
```

On peut munir ce langage avec une notion d'évaluation d'une expression vers une valeur. On la définit comme une fonction logique `eval` :

```
function eval (e: exp) : int = match e with
  | Const n → n
  | Sub e1 e2 → (eval e1) - (eval e2)
  end
```

Cette fonction est la sémantique naturelle (à grands pas) pour notre langage.

Pour définir notre relation de réduction, on commence par définir la relation $\xrightarrow{\epsilon}$, correspondant à une réduction en tête d'une expression. Pour ce langage, il y a qu'une seule règle de réduction en tête :

$$\texttt{Sub}\ (\texttt{Const}\ \texttt{v}_1)\ (\texttt{Const}\ \texttt{v}_2) \quad \xrightarrow{\epsilon} \quad \texttt{Const}\ (\texttt{v}_1 - \texttt{v}_2)$$

Pour traduire $\xrightarrow{\epsilon}$ en Why3 on introduit une fonction `head_reduction` :

```
let head_reduction (e: exp) : exp = match e with
  | Sub (Const v1) (Const v2) → Const (v1 - v2)
  | _ → absurd
  end
```

La seconde branche du filtrage représente le cas où l'expression passée en argument n'est pas un redex. Comme on souhaite appliquer `head_reduction` uniquement à des expressions qui peuvent être réduites en tête, on utilise le mot-clé `absurd` de Why3 pour marquer cette branche comme un point inaccessible du code. Pour prouver que cette branche est effectivement inatteignable [2], il faut exiger que l'argument passé à `head_reduction` soit un redex. Pour cela, on introduit le prédicat `is_redex` :

```
predicate is_redex (e: exp) = match e with
  | Sub (Const _) (Const _) → true
  | _ → false
  end
```

On peut maintenant ajouter à `head_reduction` la précondition souhaitée :

```
let head_reduction (e: exp) : exp
  requires { is_redex e }
  ...
```

Le résultat de l'appel `head_reduction e` est une expression `e'` dont sa valeur est la même que celle de `e`. On ajoute à `head_reduction` une postcondition qui spécifie exactement ce raisonnement :

```
let head_reduction (e: exp) : exp
  requires { is_redex e }
  ensures  { eval result = eval e }
  ...
```

On peut montrer facilement que la fonction `head_reduction` respecte la spécification donnée.

Pour réduire en profondeur, on introduit la règle d'inférence

$$\frac{\mathtt{e} \xrightarrow{\epsilon} \mathtt{e}'}{C[\mathtt{e}] \to C[\mathtt{e}']}$$

où C représente un contexte de réduction, qui est défini par la grammaire suivante :

$$\begin{array}{rcl} C & ::= & \square \\ & | & C[\mathtt{Sub}\ \square\ \mathtt{e}] \\ & | & C[\mathtt{Sub}\ (\mathtt{Const}\ \mathtt{v})\ \square] \end{array}$$

Le symbole $\square$ représente le trou. On définit $C[\mathtt{e}]$ comme étant le contexte C dans lequel le trou à été remplacé par l'expression `e` et on appelle cette opération *composition*. Le résultat de cette opération est une expression. La grammaire présentée induit une construction des contextes de bas en haut. Tels qu'on les a choisis, les contextes imposent ici une évaluation en appel par valeur et de gauche à droite.

Le point intéressant de cet exemple est la façon que nous allons choisir de représenter les contextes. Au lieu de représenter un contexte comme un type algébrique à trois constructeurs, on choisit ici de représenter un contexte par une fonction. Un contexte `c` est ainsi une fonction qui prend en argument une expression `e` et qui renvoie l'expression obtenue en remplaçant le trou de `c` par l'expression `e` :

```
type context = exp → exp
```

Le contexte vide, le trou, est représenté par la fonction identité :

```
let hole = fun x → x
```

Les contextes pour une réduction sur le premier ou sur le second argument de `Sub` sont représentés, respectivement, comme suit :

```
let sub_left e2 c = fun e1 → c (Sub e1 e2)
let sub_right v c = fun e1 → c (Sub (Const v) e1)
```

2. La construction `absurd` exige de prouver faux.

Cette façon de représenter les contextes nous amène à un code d'ordre supérieur élégant, plus compact qu'un code utilisant un type concret pour les contextes, tel qu'on le verra par la suite.

Les contextes étant définis, la prochaine étape pour implémenter un interprète à petits pas pour notre langage consiste à implémenter une fonction qui décompose une expression `e` en un redex `e'` et un contexte C tel que $C[\mathtt{e}'] = \mathtt{e}$. Pour notre langage cette décomposition est unique. Cette fonction utilise une fonction récursive auxiliaire `decompose_term`, dont le code Why3 est le suivant :

```
let rec decompose_term (e: exp) (c: context) : (context, exp) = match e with
  | Const _ → absurd
  | Sub (Const v1, Const v2) → (c, e)
  | Sub (Const v, e) → decompose_term e (fun x → c (Sub (Const v) x))
  | Sub (e1, e2) → decompose_term e1 (fun x → c (Sub x e2))
  end
```

Expliquons plus en détail le fonctionnement de la fonction `decompose_term`. Cette fonction prend en argument un contexte `c` et une expression `e` qu'il faut décomposer. Pour cela, on procède par une analyse par cas sur la forme de l'expression `e`. On ne peut pas décomposer une valeur, alors on utilise `absurd` pour marquer la première branche comme innatteignable. Si l'expression est de la forme `Sub (Const v1) (Const v2)`, elle est un redex. Dans ce cas-ci, on a trouvé la décomposition qu'on cherchait et la fonction renvoie ce redex et le contexte `c`. Si, en revanche, il y a dans `e` des sous-expressions qu'on peut encore réduire, alors on continue le processus de décomposition. Si la première sous-expression de `Sub` est déjà réduite, on continue la décomposition sur la seconde sous-expression avec un nouveau contexte de réduction sur la seconde sous-expression. Si la première sous-expression de `Sub` n'est pas encore réduite, la décomposition continue avec un contexte de réduction sur la première sous-expression [3].

On présente ensuite la spécification logique pour la fonction `decompose_term`. On veut appliquer `decompose_term` uniquement à des expressions qui sont pas encore des valeurs. À cet effet, on introduit une fonction logique `is_value` qui teste si une expression est une constante :

```
predicate is_value (e: exp) = match e with
  | Const _ → true
  | _ → false
  end
```

On ajoute alors un contrat à `decompose_term` et aux abstractions utilisées à l'intérieur :

```
let rec decompose_term (e: exp) (c: context) : (context, exp)
  requires { not (is_value e) }
  ensures  { let (c', e') = result in
             is_redex e' && forall res. post c e res → post c' e' res }
= match e with
  ...
  | Sub (Const v, e) →
     decompose_term e (fun x → ensures { post c (Sub (Const v) x) result } ...)
  | Sub (e1, e2) →
     decompose_term e1 (fun x → ensures { post c (Sub x e) result } ...)
  end
```

Notons qu'il n'y a pas d'effets de bord dans ce programme. En conséquence, on utilise une projection `post` à deux arguments, comme dans la section 3.1. La fonction `decompose_term` doit vérifier la propriété $C'[\mathtt{e}'] = C[\mathtt{e}]$, C' et $\mathtt{e}'$ étant le contexte et l'expression renvoyés et C et `e` les arguments. Or, comme on a choisi de représenter les contextes comme des fonctions, cette composition correspond précisément à l'application d'un contexte à son argument. Pour spécifier cela, on utilise la projection

3. Le lecteur peut remarquer qu'on a expansé le code de `hole`, `sub_left` et `sub_right` pour construire les contextes.

`post` sur un contexte et la quantification universelle pour dire que pour toute expression `res` qui vérifie `post c e res`, alors `post c' e' res` est également vérifiée. Par ailleurs, la postcondition de `decompose_term` assure (à gauche du symbole `&&`) que l'expression `e'` renvoyée est un redex.

On introduit maintenant la fonction `decompose` et sa spécification. Cette fonction prend en argument une expression et ne fait qu'appeler `decompose_term` en lui passant le contexte vide, ici représenté par la fonction identité :

```
let decompose (e: exp) : (context, exp)
  requires { not (is_value e) }
  ensures  { let (c', e') = result in is_redex e' && post c' e' e  }
= decompose_term e (fun x → x)
```

Pour évaluer une expression `e` en une valeur on introduit une fonction d'itération qui se comporte comme la clôture transitive de la relation $\rightarrow$. Son code Why3 est le suivant :

```
let rec red (e: exp) : int = match e with
  | Const v → v
  | _ →
    let (c, r) = decompose e in
    let     r' = head_reduction r in
    red (c r')
  end
```

Si l'expression `e` est de la forme `Const v`, elle est en forme normale et elle ne peut plus être réduite. Si, en revanche, `e` n'est pas encore une constante on (1) décompose cette expression en le redex `r` et le contexte `c` ; (2) on réduit `r` en tête et on obtient l'expression `r'` ; (3) on continue l'itération avec l'expression obtenue par la composition de `c` et `r'`. La composition se matérialise, d'une façon élégante, comme l'application de `c` à `r'`. La valeur renvoyée par l'appel `red e` doit être la même que celle renvoyée par une évaluation à grands pas. On munit donc `red` du contrat suivant :

```
let rec red (e: exp) : int
  ensures { result = eval e }
```

On procède à la défonctionnalisation de cet exemple pour montrer sa correction avec Why3. On commence par introduire le type `context` défonctionnalisé :

```
type context = CHole | CApp_left context exp | CApp_right int context
```

À partir des abstractions présentes dans le programme de départ, on engendre la fonction `apply` conjointement avec sa spécification :

```
let rec apply (c: context) (arg: exp) : exp
  ensures { post c arg result }
= match c with
  | CHole → let x = arg in x
  | CApp_left c e → let x = arg in apply c (Sub x e)
  | CApp_right v c → let x = arg in apply c (Sub (Const v) x)
  end
```

où le prédicat `post` peut être déduit des postconditions également présentes dans le programme de départ :

```
predicate post (c: context) (arg result: exp) = match c with
  | CHole → let x = arg in x = result
  | CApp_left c e → let x = arg in post c (Sub x e) result
  | CApp_right v c → let x = arg in post c (Sub (Const v) x) result
  end
```

Pour obtenir le programme défonctionnalisé final, il suffit, comme montré pour les exemples précédents, de remplacer toutes les applications des abstractions par des appels à `apply` et toutes les occurrences de ces mêmes abstractions par le constructeur respectif du type `context` défonctionnalisé.

Si on donnait le programme défonctionnalisé et sa spécification à Why3 on arriverait à prouver toutes les obligations de preuve engendrées, sauf la postcondition de la fonction `red`. Pour aider les démonstrateurs à prouver cette postcondition, on introduit le lemme suivant :

```
lemma post_eval: forall c arg1 arg2 r1 r2.
  eval arg1 = eval arg2 → post c arg1 r1 → post c arg2 r2 → eval r1 = eval r2
```

Ce lemme exprime que pour toutes expressions `arg1` et `arg2` dont l'évaluation produit la même valeur, alors les expressions obtenues par la composition de `arg1` et `arg2` avec le même contexte `c` doivent s'évaluer en la même valeur. On prouve ce résultat par récurrence sur `c` et on écrit pour cela une *lemma function*, c'est-à-dire, un programme fantôme qui termine et qui n'a pas d'effet de bord observable, dont le contrat sera automatiquement traduit en l'énoncé donné plus haut :

```
let rec lemma post_eval (c: context) (arg1 arg2 r1 r2: exp)
  requires { eval arg1 = eval arg2 }
  requires { post c arg1 r1 && post c arg2 r2 }
  ensures  { eval r1 = eval r2 }
  variant  { c }
= match c with
  | CHole → ()
  | CApp_left c e → post_eval c (Sub arg1 e) (Sub arg2 e) r1 r2
  | CApp_right n c → post_eval c (Sub (Const n) arg1) (Sub (Const n) arg2) r1 r2
  end
```

Les appels récursifs de ce programme simulent l'application des hypothèses de récurrence. Avec ce lemme auxiliaire, la postcondition de `red` est prouvée automatiquement par des démonstrateurs SMTs. L'énoncé de ce lemme quantifie universellement sur les valeurs du type `context`. Écrire un tel énoncé directement dans le programme d'ordre supérieur pourrait introduire une contradiction, vu qu'on ne peut pas affirmer ce résultat pour n'importe quelle fonction du type `exp` → `exp`. Ce serait, néanmoins, intéressant d'avoir un moyen d'écrire ce genre d'énoncés dans le code de départ (en contraignant les fonctions abstraites acceptées) et les traduire automatiquement pour le code défonctionnalisé.

Notons que le type des contextes défonctionnalisés correspond à un *zipper* [10] pour les expressions de notre langage. Cette façon de représenter les contextes nous permettrait de mettre en œuvre une fonction d'évaluation efficace. Une telle fonction est normalement appelée *refocusing* [5]. En utilisant des contextes comme des fonctions, nos opérations de décomposition et composition ont une complexité linéaire dans le pire des cas, ce qui donne à la fonction `red` un coût potentiellement quadratique en la taille d'une expression.

4. Discussion et perspectives

Dans cet article nous avons exploré l'utilisation de la défonctionnalisation comme une technique de preuve pour des programmes d'ordre supérieur contenant potentiellement des effets. L'idée est d'annoter directement un programme d'ordre supérieur et ensuite de défonctionnaliser ce programme et sa spécification vers un programme du premier ordre. Un outil de vérification déductive existant peut être alors utilisé pour montrer la correction de ce programme. Cette façon de procéder nous permet d'utiliser des outils de vérification déductive de programmes du premier ordre existants, sans avoir besoin de leur ajouter le support de l'ordre supérieur (ce qui impliquerait, très probablement, un changement profond dans les noyaux de ces outils, tel que leur langage de programmation, leur logique et leur calcul d'obligations de preuve). Nous avons illustré cette approche à l'aide de trois

exemples que nous avons (manuellement) défonctionnalisé. Les programmes du premier ordre obtenus ont été automatiquement vérifiés dans le système Why3. L'utilisation de la défonctionnalisation dans un contexte de preuve est, à notre connaissance, nouvelle.

La défonctionnalisation est généralement considérée comme une technique globale de transformation de programme, c'est-à-dire qu'on doit connaître toutes les fonctions utilisées comme des valeurs de première classe dans un programme au moment d'appliquer cette transformation. Cela implique qu'on ne peut pas espérer prouver tous les programmes d'ordre supérieur en les défonctionnalisant. Néanmoins, il semble que la méthode proposée peut être appliquée avec succès au moins aux programmes écrits en style CPS, vu qu'ils s'agit de programmes dont on connaît toutes les abstractions utilisées dans les fonctions d'ordre supérieur.

Perspectives. Le travail présenté dans cet article reste, pour l'instant, exploratoire. On a utilisé la défonctionnalisation pour prouver plusieurs exemples de programmes d'ordre supérieur et les résultats nous encouragent à suivre l'exploration de cette méthodologie. Notre objectif est d'améliorer et de formaliser notre utilisation de la défonctionnalisation sur des programmes d'ordre supérieur. Nous commencerons par formaliser la classe de programmes pour lesquels cette technique peut être appliqué, afin qu'on puisse comprendre si la défonctionnalisation peut être utilisée comme une méthode robuste de preuve de programmes d'ordre supérieur contenant des effets.

La fonction `apply`, engendrée pendant la défonctionnalisation, simule l'application d'une fonction (représentée par une valeur du type algébrique obtenue par défonctionnalisation) à son argument. La fonction `apply` procède par une analyse par cas sur chaque fonction trouvée dans le programme de départ, où chaque branche correspond à la définition de l'abstraction correspondante. Ainsi, chacune de ces branches doit renvoyer une valeur du même type, ce qui est vrai uniquement lorsque toutes les abstractions du programme ont le même type $\tau_1 \to \tau_2$. Pour régler ce problème, Pottier et Gauthier [16] ont proposé l'utilisation des *types algébriques généralisés* (de l'anglais *generalized algebraic data types*, GADT) pour défonctionnaliser un programme avec des abstractions de différents types. Cette solution nous intéresse et nous étudions des moyens possibles pour étendre le langage et la logique du système Why3 pour ajouter le support des GADT. Pour l'instant, nous introduisons plusieurs fonctions `apply` si jamais le programme initial possède des abstractions de plusieurs types différents.

Pour donner des garanties fortes sur la fiabilité d'un programme, sa preuve de terminaison joue un rôle fondamental. Dans cet article on a mentionné comment on pourrait montrer que le programme défonctionnalisé termine, mais au prix d'une intervention sur le programme généré. Il serait plutôt souhaitable de pouvoir spécifier directement la terminaison dans le programme d'ordre supérieur avec des mesures de décroissance appropriées, qui seraient traduites vers le programme défonctionnalisé.

Les itérateurs sont des exemples de fonctions d'ordre supérieur très souvent utilisés pour réaliser le parcours des éléments d'une structure de données. Dans des travaux précédents [8, 9], nous avons proposé des techniques pour prouver la correction de programmes qui mettent en œuvre de tels itérateurs et des programmes qui les utilisent. À cet effet, nous avons commencé le développement d'un outil qui prend en entrée des programmes d'ordre supérieur annotés et qui se servait de Why3 pour engendrer des obligations de preuve. Nous envisageons d'étendre cet outil avec le support pour la défonctionnalisation et d'augmenter ainsi l'ensemble des programmes d'ordre supérieur acceptés par l'outil.

Finalement, pour mieux évaluer son utilité, nous souhaitons appliquer la preuve par défonctionnalisation sur un exemple plus complexe. Un bon candidat est l'algorithme de Koda-Ruskey [13] pour l'énumération de tous les idéaux d'un ensemble partiellement ordonné. Une implantation d'ordre supérieur existe pour cet algorithme [6] et nous envisageons de reprendre ce code et prouver sa correction à l'aide de notre méthodologie.

Remerciements. Je remercie Lucas Baudin, Richard Bonichon, Martin Clochard et Léon Gondelman pour leurs remarques et suggestions pendant la préparation de cet article. Je tiens à remercier très chaleureusement Jean-Christophe Filliâtre. Ses conseils, ses relectures attentives et ses corrections ont été précieux pour l'achèvement de ce travail.

Références

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003.

[2] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[3] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8) :534–549, 2009. Special Issue on Mathematics of Program Construction (MPC 2006).

[4] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174. ACM Press, 2001.

[5] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. *Electr. Notes Theor. Comput. Sci.*, 59(4) :358–374, 2001.

[6] Jean-Christophe Filliâtre. La supériorité de l'ordre supérieur. In *Journées Francophones des Langages Applicatifs*, pages 15–26, Anglet, France, January 2002.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[8] Jean-Christophe Filliâtre and Mário Pereira. Itérer avec confiance. In *Vingt-septièmes Journées Francophones des Langages Applicatifs*, Saint-Malo, France, January 2016.

[9] Jean-Christophe Filliâtre and Mário Pereira. A modular way to reason about iteration. In *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, Minneapolis, MN, USA, June 2016. Springer.

[10] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, September 1997.

[11] Johannes Kanig. *Spécification et preuve de programmes d'ordre supérieur*. Thèse de doctorat, Université Paris-Sud, 2010.

[12] Johannes Kanig and Jean-Christophe Filliâtre. Who : A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, August 2009.

[13] Yasunori Koda and Frank Ruskey. A Gray Code for the Ideals of a Forest Poset. *Journal of Algorithms*, (15) :324–340, 1993.

[14] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot : Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

[15] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2) :125–159, 1975.

[16] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19 :125–162, March 2006.

[17] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, pages 305–335, 2008.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4) :363–397, December 1998.